1 TableGen Programmer’s Guide

- 1.1 Introduction
  - 1.1.1 Concepts
- 1.2 Source Files
- 1.3 Lexical Analysis
  - 1.3.1 Literals
  - 1.3.2 Identifiers
  - 1.3.3 Bang operators
  - 1.3.4 Include files
- 1.4 Types
- 1.5 Values and Expressions
  - 1.5.1 Simple values
  - 1.5.2 Suffixed values
- 1.6 Statements
  - 1.6.1 `class` — define an abstract record class
    - 1.6.1.1 Record Bodies
  - 1.6.2 `def` — define a record
  - 1.6.3 Interlude 1: Class and record examples
  - 1.6.4 `let` — override fields in classes or records
  - 1.6.5 `multiclass` — define multiple records
  - 1.6.6 `defm` — define multiple records
  - 1.6.7 Interlude 2: Multiclass and defm examples
  - 1.6.8 `defset` — create a definition set
  - 1.6.9 `defvar` — define a variable
  - 1.6.10 `foreach` — iterate over a sequence
  - 1.6.11 `if` — select statements based on a test
- 1.7 Additional Details
  - 1.7.1 Defvar in record body
  - 1.7.2 How records are built
- 1.8 Preprocessing Facilities
- 1.9 Appendix A: Bang Operators
- 1.10 Appendix B: Sample Record

1.1 Introduction

This document describes the LLVM TableGen facility in complete detail. It is intended for the programmer who is using TableGen to produce tables for a project. If you are looking for a simple overview, check out Welcome to LLVM's documentation!.

The purpose of TableGen is to generate complex output files based on information from input files that are significantly easier to code than the output files are, and also easier to maintain and modify over time. The information is coded in the form of typed records that are processed by TableGen and then passed on to various backends. Each backend extracts information from a subset of the records and generates one or more output files. These output files are typically `.inc` files for C++, but may be any sort of file that the backend developer needs.

An example of a backend is `RegisterInfo`, which generates the register file information for a particular target machine, for use by the LLVM target-independent code generator. See BackEnds for a description of the LLVM TableGen backends. Here are a few of the things backends can do.

- Generate the register file information for a particular target machine.
- Generate the instruction definitions for a target.
- Generate the patterns that the code generator uses to match instructions to IR nodes.
- Generate semantic attribute identifiers for Clang.
- Generate AST declaration node definitions for Clang.
- Generate AST statement node definitions for Clang.

1.1.1 Concepts

TableGen source files contain two primary items: *abstract records* and *concrete records*. In this and other TableGen documents, abstract records are called *classes*. In addition, concrete records are usually just called records, although sometimes the term *record* refers to both classes and concrete records.

Classes and concrete records have a unique *name*. Associated with that name is a list of *fields* with values and an optional list of *superclasses* (sometimes called parent classes). The fields are the primary data that backends will process. Note that TableGen assigns no meanings to fields; the meanings are entirely up to the backends and the programs that incorporate the output of those backends.

All the fields in a concrete record usually have values. A backend processes some subset of the concrete records built by the TableGen parser and emits the output files. In a complex program such as LLVM, there can be many concrete records and some of them can have an unexpectedly large number of fields.

In order to reduce the complexity of TableGen files, classes are used to abstract out groups of record fields. For example, a few classes may abstract the concept of a machine register file, while other classes may abstract the instruction formats, and still others may abstract the individual instructions. TableGen allows an arbitrary hierarchy of classes, so that the abstraction classes for two concepts can share a third parent class that abstracts common "sub-concepts" from the two original concepts.

In order to make classes more useful, a concrete record (or another class) can request a class as a superclass and pass *template arguments* to it. These template arguments can be used in the fields of the superclass to initialize them in a custom manner. That is, record or class `A` can request superclass `S` with one set of template arguments, while record or class `B` can request `S` with a different set of arguments.

Without template arguments, many more classes would be required, one for each combination of the template arguments.

TableGen provides *multiclasses* to collect a group of record definitions in one place. A multiclass is a sort of macro that can be "invoked" to define multiple concrete records all at once. A multiclass can inherit from other multiclasses, which means that the base multiclass inherits all the definitions from the parent multiclass.

Appendix B: Sample Record illustrates a complex record in the Intel x86 target and the simple way in which it is defined.

1.2 Source Files

TableGen source files are plain ASCII text files. The files can contain statements, comments, and blank lines (see Lexical Analysis). The standard file extension for TableGen files is `.td`.

TableGen files can grow quite large, so there is an include mechanism that allows one file to include the content of another file (see Include Files). This allows large files to be broken up into smaller ones, and also provides a simple library mechanism where multiple source files can include the same library file.

1.3 Lexical Analysis

The lexical and syntax notation used here is intended to imitate Python's notation. In particular, for lexical definitions, the productions operate at the character level and there is no implied whitespace between elements. The syntax definitions operate at the token level, so there is implied whitespace between tokens.

TableGen supports BCPL-style comments (`// ...`) and nestable C-style comments (`/* ... */`). TableGen also provides simple Preprocessing Facilities.

Formfeed characters may be used freely in files to produce page breaks when the file is printed for review.

The following are the basic punctuation tokens:

```
- + [ ] { } ( ) < > : ; .  = ? #
```

1.3.1 Literals

Numeric literals take one of the following forms:

```
TokInteger     ::=  DecimalInteger | HexInteger | BinInteger
DecimalInteger ::=  ["+" | "-"] ("0"..."9")+
HexInteger     ::=  "0x" ("0"..."9" | "a"..."f" | "A"..."F")+
BinInteger     ::=  "0b" ("0" | "1")+
```

Observe that the **`DecimalInteger`** token includes the optional `+` or `-` sign, unlike most languages where the sign would be treated as a unary operator.

Note that **BinInteger** creates of value of type `bits<n>`, where *n* is the number of bits specified. The bits are not extended or truncated to any particular word size. This value will be implicitly converted to an integer where necessary.

TableGen has two kinds of string literals:

```
TokString       ::=  '"' (non-'"' characters and escapes) '"'
TokCodeFragment ::=  "[{" (shortest text not containing "}]") "}]"
```

A **TokCodeFragment** is nothing more than a multi-line string literal delimited by `[{` and `}]`. It can break across lines.

The current implementation accepts the following escape sequences:

```
\\ \' \" \t \n
```

1.3.2 Identifiers

TableGen has name- and identifier-like tokens, which are case-sensitive.

```
ualpha        ::=  "a"..."z" | "A"..."Z" | "_"
TokIdentifier ::=  ("0"..."9")* ualpha (ualpha | "0"..."9")*
TokVarName    ::=  "$" ualpha (ualpha |  "0"..."9")*
```

Note that, unlike most languages, TableGen allows **TokIdentifier** to begin with an integer. In case of ambiguity, a token is interpreted as a numeric literal rather than an identifier.

TableGen has the following reserved words, which cannot be used as identifiers:

```
bit        bits        class        code         dag
def        else        foreach      defm         defset
defvar     field       if           in           include
int        let         list         multiclass   string
then
```

Warning:

The `field` reserved word is deprecated.

1.3.3 Bang operators

TableGen provides "bang operators" that have a wide variety of uses:

```
BangOperator ::=  one of
```

```
!add     !and          !cast         !con         !dag
!empty   !eq           !foldl        !foreach     !ge
!getop   !gt           !head         !if          !isa
!le      !listconcat   !listsplat    !lt          !mul
!ne      !or           !setop        !shl         !size
!sra     !srl          !strconcat    !subst       !tail
```

The `!cond` operator has a slightly different syntax compared to other bang operators, so it is defined separately:

```
CondOperator ::=  !cond
```

See Appendix A: Bang Operators for a description of each bang operator.

1.3.4 Include files

TableGen has an include mechanism. The content of the included file lexically “replaces” the `include` directive and is then parsed just like the main file.

```
IncludeDirective ::=  "include" TokString
```

1.4 Types

The TableGen language is statically typed, using a simple but complete type system. Types are used to check for errors, to perform implicit conversions, and to help interface designers constrain the allowed input. Every value is required to have an associated type.

TableGen supports a mixture of low-level types (e.g., `bit`) and high-level types (e.g., `dag`). This flexibility allows you to describe a wide range of records conveniently and compactly.

```
Type    ::=  "bit" | "int" | "string" | "code" | "dag"
            | "bits" "<" TokInteger ">"
            | "list" "<" Type ">"
            | ClassID
ClassID ::=  TokIdentifier
```

`bit`
A `bit` is a boolean value that can be 0 or 1.

`int`
The `int` type represents a simple 64-bit integer value, such as 5 or -42.

`string`
The `string` type represents an ordered sequence of characters of arbitrary length.

`code`
The `code` type represents a code fragment. The values are the same as those for the `string` type; the

code type is provided just to indicate programmer intention.

`bits<`*n*`>`
The `bits` type is a fixed-size integer of arbitrary length *n* that is treated as individual bits. This type is useful because it can handle some bits being defined while others are undefined.

`list<`*type*`>`
This type represents a list whose elements are of the *type* specified in angle brackets. The element type is arbitrary; it can even be another list type. List elements are indexed from 0.

`dag`
This type represents a nestable directed acyclic graph (DAG) of elements.

`ClassID`
Specifying a class name in a type context indicates that the type of the defined value must be a subclass of the specified class. This is useful in conjunction with the `list` type; for example, to constrain the elements of the list to a common base class (e.g., a `list<Register>` can only contain definitions derived from the `Register` class). The **`ClassID`** must name a class that has been previously declared or defined.

To date, these types have been sufficient for describing the data that TableGen generates, but it is straightforward to extend this list if needed.

1.5 Values and Expressions

There are many contexts in TableGen statements where a value is required. A common example is in the definition of a record, where each field is specified by a name and value. TableGen allows for a reasonable number of different forms when building up values. These forms allow the TableGen file to be written in a syntax that is natural for the application.

Note that all of the values have rules for converting them from one type to another. For example, these rules allow you to assign a value like `7` to an entity of type `bits<4>`.

```
Value       ::=  SimpleValue ValueSuffix*
ValueSuffix ::=  "{" RangeList "}"
               | "[" RangeList "]"
               | "." TokIdentifier
RangeList   ::=  RangePiece ("," RangePiece)*
RangePiece  ::=  TokInteger
               | TokInteger "-" TokInteger
               | TokInteger TokInteger
```

The peculiar last form of **`RangePiece`** is due to the fact that the “`-`” is included in the **`TokInteger`**, hence `1-5` gets lexed as two consecutive **`TokInteger`s, with values ``1`** and `-5`, instead of `1`, `-`, and `5`. The **`RangeList`** can be thought of as specifying “list slice” in some contexts.

1.5.1 Simple values

The **SimpleValue** has a number of forms.

```
SimpleValue ::=  TokInteger | TokString+ | TokCodeFragment
```

A value can be an integer literal, a string literal, or a code fragment literal. Multiple adjacent string literals are concatenated as in C/C++; the single value is the concatenation of the strings. Code fragments become strings and then are indistinguishable from them.

```
SimpleValue ::=  "?"
```

A question mark represents an uninitialized value.

```
SimpleValue ::=  "{" [ValueList] "}"
ValueList   ::=  ValueListNE
ValueListNE ::=  Value ("," Value)*
```

This value represents a sequence of bits, which can be used to initialize a `bits<`*n*`>` field (note the braces). When doing so, the values must represent a total of *n* bits.

```
SimpleValue ::=  "[" ValueList "]" ["<" Type ">"]
```

This value is a list initializer (note the brackets). The values in brackets are the elements of the list. The optional **Type** can be used to indicate a specific element type; otherwise the element type will be inferred from the given values. TableGen can usually infer the type, although sometimes not when the value is the empty list (`[]`).

```
SimpleValue ::=  "(" DagArg [DagArgList] ")"
DagArgList  ::=  DagArg ("," DagArg)*
DagArg      ::=  Value [":" TokVarName] | TokVarName
```

This represents a DAG initializer (note the parentheses). The first **DagArg** is called the "operator" of the DAG.

```
SimpleValue ::=  TokIdentifier
```

The resulting value is the value of the entity named by the identifier. The possible identifiers are described here, but the descriptions will make more sense after reading the remainder of this guide.

- A template argument of a `class`, such as the use of `Bar` in:

  ```
  class Foo <int Bar> {
    int Baz = Bar;
  }
  ```

- The implicit template argument `NAME` in a `class` or `multiclass` definition.

- A field local to a `class`, such as the use of `Bar` in:

```
class Foo {
  int Bar = 5;
  int Baz = Bar;
}
```

- The name of a record definition, such as the use of `Bar` in the definition of `Foo`:

```
def Bar : SomeClass {
  int X = 5;
}

def Foo {
  SomeClass Baz = Bar;
}
```

- A field local to a record definition, such as the use of `Bar` in:

```
def Foo {
  int Bar = 5;
  int Baz = Bar;
}
```

  Fields defined in the record's superclasses can be accessed the same way.

- A template argument of a `multiclass`, such as the use of `Bar` in:

```
multiclass Foo <int Bar> {
  def : SomeClass<Bar>;
}
```

- A variable defined with the `defvar` or `defset` statements.

- The iteration variable of a `foreach`, such as the use of `i` in:

```
foreach i = 0-5 in
  def Foo#i;
```

```
SimpleValue ::=  ClassID "<" ValueListNE ">"
```

This form creates a new anonymous record definition (as would be created by an unnamed `def` inheriting from the given class with the given template arguments; see def) and the value is that record. (A field of the record can be obtained using a suffix; see Suffixed Values.)

```
SimpleValue ::=  BangOperator ["<" Type ">"] "(" ValueListNE ")"
                | CondOperator "(" CondClause ("," CondClause)* ")"
CondClause  ::=  Value ":" Value
```

The bang operators provide functions that are not available with the other **`SimpleValue`** values. Except in the case of `!cond`, a bang operator takes a list of arguments enclosed in parentheses and performs some function on those arguments, producing a value for that bang operator. The `!cond` operator takes a list of pairs of arguments separated by colons. See Appendix A: Bang Operators for a description of each bang operator.

1.5.2 Suffixed values

The **`SimpleValue`** values described above can be specified with certain suffixes. The purpose of a suffix is to obtain a subvalue of the primary value. Here are the possible suffixes for some primary *value*.

value`{17}`
: The final value is bit 17 of the integer *value* (note the braces).

value`{8-15}`
: The final value is bits 8–15 of the integer *value*. The order of the bits can be reversed by specifying `{15-8}`.

value`[4-7,17,2-3,4]`
: The final value is a new list that is a slice of the list *value* (note the brackets). The new list contains elements 4, 5, 6, 7, 17, 2, 3, and 4. Elements may be included multiple times and in any order.

value`.` *field*
: The final value is the value of the specified *field* in the specified record *value*.

1.6 Statements

The following statements may appear at the top level of TableGen source files.

```
TableGenFile ::=  Statement*
Statement    ::=  Class | Def | Defm | Defset | Defvar | Foreach
                 | If | Let | MultiClass
```

The following sections describe each of these top-level statements.

1.6.1 `class` — define an abstract record class

A `class` statement defines an abstract record class from which other classes and records can inherit.

```
Class           ::=  "class" ClassID [TemplateArgList] RecordBody
TemplateArgList ::=  "<" TemplateArgDecl ("," TemplateArgDecl)* ">"
TemplateArgDecl ::=  Type TokIdentifier ["=" Value]
```

A class can be parameterized by a list of “template arguments,” whose values can be used in the class’s record body. These template arguments are specified each time the class is inherited by another class or record.

If a template argument is not assigned a default value with `=`, it is uninitialized (has the “value” `?`) and must be specified in the template argument list when inherited. If an argument is assigned a default value, then it need not be specified in the argument list. The template argument default values are evaluated from left to right.

The **`RecordBody`** is defined below. It can include a list of parent classes from which the current class inherits, along with field definitions and other statements. When a class `C` inherits from another class `D`, the fields of `D` are effectively merged into the fields of `C`.

A given class can only be defined once. A `class` statement is considered to define the class if *any* of the following are true (the **`RecordBody`** elements are described below).

- The **`TemplateArgList`** is present, or
- The **`BaseClassList`** in the **`RecordBody`** is present, or
- The **`Body`** in the **`RecordBody`** is present and not empty.

You can declare an empty class by specifying an empty **`TemplateArgList`** and an empty **`RecordBody`**. This can serve as a restricted form of forward declaration. Note that records derived from a forward-declared class will inherit no fields from it, because those records are built when their declarations are parsed, and thus before the class is finally defined.

Every class has an implicit template argument named `NAME`, which is bound to the name of the **`Def`** or **`Defm`** inheriting the class. The value of `NAME` is undefined if the class is inherited by an anonymous record.

See Interlude 1: Class and Record Examples for examples.

1.6.1.1 Record Bodies

Record bodies appear in both class and record definitions. A record body can include a base class list, which specifies the classes from which the current class or record inherit fields. Such classes are called the *superclasses* of the class or record. The record body also includes the main body of the definition, which contains the specification of the fields of the class or record.

```
RecordBody      ::=  BaseClassList Body
BaseClassList   ::=  [":" BaseClassListNE]
BaseClassListNE ::=  ClassRef ("," ClassRef)*
ClassRef        ::=  (ClassID | MultiClassID) ["<" ValueList ">"]
```

A **`BaseClassList`** containing a **`MultiClassID`** is valid only in the class list of a `defm` statement. In that case, the ID must be the name of a multiclass.

```
Body     ::=  ";" | "{" BodyItem* "}"
BodyItem ::=  Type TokIdentifier ["=" Value] ";"
             | "let" TokIdentifier ["{" RangeList "}"] "=" Value ";"
             | "defvar" TokIdentifier "=" Value ";"
```

A field definition in the body specifies a field to be included in the class or record. If no initial value is specified, then the field's value is uninitialized. The type must be specified; TableGen will not infer it from the value.

The `let` form is used to reset a field to a new value. This can be done for fields defined directly in the body or fields inherited from superclasses. A **`RangeList`** can be specified to reset certain bits in a `bit<n>` field.

The `defvar` form defines a variable whose value can be used in other value expressions within the body. The variable is not a field: it does not become a field of the class or record being defined. Variables are provided to hold temporary values while processing the body. See Defvar in Record Body for more details.

When class `C2` inherits from class `C1`, it acquires all the field definitions of `C1`. As those definitions are merged into class `C2`, any template arguments passed to `C1` by `C2` are substituted into the definitions. In other words, the abstract record fields defined by `C1` are expanded with the template arguments before being merged into `C2`.

1.6.2 `def` — define a record

A `def` statement defines a record.

```
Def       ::=  "def" [NameValue] RecordBody
NameValue ::=  Value
```

The name value is optional. If specified, it is parsed in a special mode where undefined (unrecognized) identifiers are interpreted as literal strings. In particular, global identifiers are considered unrecognized. These include global variables defined by `defvar` and `defset`.

If no name value is given, the record is *anonymous*. The final name of an anonymous record is unspecified but globally unique.

Special handling occurs if a `def` appears inside a `multiclass` statement. See the `multiclass` section below for details.

A record can inherit from one or more classes by specifying the **`BaseClassList`** clause at the beginning of its record body. All of the fields in the base classes are added to the record. If two or more base classes provide the same field, the record ends up with the field value of the last base class.

As a special case, the name of a record can be passed in a template argument to that record's superclasses. For example:

```
class A <dag d> {
  dag the_dag = d;
}

def rec1 : A<(ops rec1)>
```

The DAG `(ops rec1)` is passed as a template argument to class `A`. Notice that the DAG includes `rec1`, the record being defined.

The steps taken to create a new record are somewhat complex. See How records are built.

See Interlude 1: Class and Record Examples for examples.

1.6.3 Interlude 1: Class and record examples

Here is a simple TableGen file with one class and two record definitions.

```
class C {
  bit V = 1;
}

def X : C;
def Y : C {
  let V = 0;
  string Greeting = "Hello!";
}
```

First, the abstract class `C` is defined. It has one field named `V` that is a bit initialized to 1.

Next, two records are defined, derived from class `C`; that is, with `C` as their superclass. Thus they both inherit the `V` field. Record `Y` also defines another string field, `Greeting`, which is initialized to `"Hello!"`. In addition, `Y` overrides the inherited `V` field, setting it to 0.

A class is useful for isolating the common features of multiple records in one place. A class can initialize common fields to default values, but records inheriting from that class can override the defaults.

TableGen supports the definition of parameterized classes as well as nonparameterized ones. Parameterized classes specify a list of variable declarations, which may optionally have defaults, that are bound when the class is specified as a superclass of another class or record.

```
class FPFormat <bits<3> val> {
  bits<3> Value = val;
}

def NotFP      : FPFormat<0>;
def ZeroArgFP  : FPFormat<1>;
def OneArgFP   : FPFormat<2>;
def OneArgFPRW : FPFormat<3>;
def TwoArgFP   : FPFormat<4>;
def CompareFP  : FPFormat<5>;
def CondMovFP  : FPFormat<6>;
def SpecialFP  : FPFormat<7>;
```

The purpose of the `FPFormat` class is to act as a sort of enumerated type. It provides a single field, `Value`, which holds a 3-bit number. Its template argument, `val`, is used to set the `Value` field. Each of the eight records is defined with `FPFormat` as its superclass. The enumeration value is passed in angle brackets as the template argument. Each record will inherent the `Value` field with the appropriate enumeration value.

Here is a more complex example of classes with template arguments. First, we define a class similar to the `FPFormat` class above. It takes a template argument and uses it to initialize a field named `Value`. Then we define four records that inherit the `Value` field with its four different integer values.

```
class ModRefVal <bits<2> val> {
  bits<2> Value = val;
}

def None   : ModRefVal<0>;
def Mod    : ModRefVal<1>;
def Ref    : ModRefVal<2>;
def ModRef : ModRefVal<3>;
```

This is somewhat contrived, but let's say we would like to examine the two bits of the `Value` field independently. We can define a class that accepts a `ModRefVal` record as a template argument and splits up its value into two fields, one bit each. Then we can define records that inherit from `ModRefBits` and so acquire two fields from it, one for each bit in the `ModRefVal` record passed as the template argument.

```
class ModRefBits <ModRefVal mrv> {
  // Break the value up into its bits, which can provide a nice
  // interface to the ModRefVal values.
  bit isMod = mrv.Value{0};
  bit isRef = mrv.Value{1};
}

// Example uses.
def foo   : ModRefBits<Mod>;
def bar   : ModRefBits<Ref>;
def snork : ModRefBits<ModRef>;
```

This illustrates how one class can be defined to reorganize the fields in another class, thus hiding the internal representation of that other class.

Running `llvm-tblgen` on the example prints the following definitions:

```
def bar {      // Value
  bit isMod = 0;
  bit isRef = 1;
}
def foo {      // Value
  bit isMod = 1;
  bit isRef = 0;
}
def snork {      // Value
  bit isMod = 1;
  bit isRef = 1;
}
```

1.6.4 `let` — override fields in classes or records

A `let` statement collects a set of field values (sometimes called *bindings*) and applies them to all the classes and records defined by statements within the scope of the `let`.

```
Let     ::=   "let" LetList "in" "{" Statement* "}"
            | "let" LetList "in" Statement
LetList ::=  LetItem ("," LetItem)*
LetItem ::=  TokIdentifier ["<" RangeList ">"] "=" Value
```

The `let` statement establishes a scope, which is a sequence of statements in braces or a single statement with no braces. The bindings in the **`LetList`** apply to the statements in that scope.

The field names in the **`LetList`** must name fields in classes inherited by the classes and records defined in the statements. The field values are applied to the classes and records *after* the records inherit all the fields from their superclasses. So the `let` acts to override inherited field values.

Top-level `let` statements are often useful when a few fields need to be overriden in several records. Here are two examples. Note that `let` statements can be nested.

```
let isTerminator = 1, isReturn = 1, isBarrier = 1, hasCtrlDep = 1 in
  def RET : I<0xC3, RawFrm, (outs), (ins), "ret", [(X86retflag 0)]>;

let isCall = 1 in
  // All calls clobber the non-callee saved registers...
  let Defs = [EAX, ECX, EDX, FP0, FP1, FP2, FP3, FP4, FP5, FP6, ST0,
              MM0, MM1, MM2, MM3, MM4, MM5, MM6, MM7, XMM0, XMM1, XMM2,
              XMM3, XMM4, XMM5, XMM6, XMM7, EFLAGS] in {
    def CALLpcrel32 : Ii32<0xE8, RawFrm, (outs), (ins i32imm:$dst, variable_ops),
                           "call\t${dst:call}", []>;
    def CALL32r     : I<0xFF, MRM2r, (outs), (ins GR32:$dst, variable_ops),
                        "call\t{*}$dst", [(X86call GR32:$dst)]>;
    def CALL32m     : I<0xFF, MRM2m, (outs), (ins i32mem:$dst, variable_ops),
                        "call\t{*}$dst", []>;
  }
```

Note that a top-level `let` will not override fields defined in the classes or records themselves.

1.6.5 `multiclass` — define multiple records

While classes with template arguments are a good way to factor out commonality between multiple records, multiclasses allow a convenient method for defining multiple records at once. For example, consider a 3-address instruction architecture whose instructions come in two formats: `reg = reg op reg` and `reg = reg op imm` (e.g., SPARC). We would like to specify in one place that these two common formats exist, then in a separate place specify what all the operations are. The `multiclass` and `defm` statements accomplish this goal. You can think of a multiclass as a macro that expands into multiple records.

```
MultiClass          ::=  "multiclass" TokIdentifier [TemplateArgList]
                         [":" BaseMultiClassList]
                         "{" Statement+ "}"
```

```
BaseMultiClassList ::=  MultiClassID ("," MultiClassID)*
MultiClassID       ::=  TokIdentifier
```

As with regular classes, the multiclass has a name and can accept template arguments. The body of the multiclass contains a series of statements that define records, using **`Def`** and **`Defm`**. In addition, **`Defvar`**, **`Foreach`**, and **`Let`** statements can be used to factor out even more common elements.

When a named (non-anonymous) record is defined in a multiclass and the record's name does not contain a use of the implicit template argument `NAME` (see NAME), such a use is automatically prepended to the name. That is, the following are equivalent inside a multiclass:

```
def Foo ...
def NAME#Foo ...
```

The records defined in a multiclass are instantiated when the multiclass is "invoked" by a `defm` statement outside the multiclass definition. Each `def` statement produces a record. As with top-level `def` statements, these definitions can inherit from multiple superclasses.

See Interlude 2: Multiclass and Defm Examples for examples.

1.6.6 `defm` — define multiple records

Once multiclasses have been defined, you use the `defm` statement to instantiate the multiple records specified in those multiclasses.

```
Defm ::=  "defm" [NameValue] BaseClassList ";"
```

The optional **`NameValue`** is formed in the same way as the name of a `def`. The **`BaseClassList`** is a colon followed by a list of at least one multiclass and any number of regular classes. The multiclasses must precede the regular classes.

This statement instantiates all the records defined in all the specified multiclasses. Those records also receive the fields defined in the specified regular classes.

The name is parsed in the same special mode used by `def`. If the name is not included, a globally unique name is used. That is, the following examples end up with different names:

```
defm    : SomeMultiClass<...>;   // A globally unique name.
defm "" : SomeMultiClass<...>;   // An empty name.
```

The `defm` statement can be used in a multiclass body. When this occurs, the second variant is equivalent to:

```
defm NAME : SomeMultiClass<...>;
```

More generally, when `defm` occurs in a multiclass and its name does not include a use of the implicit template argument `NAME`, then `NAME` will be prepended automatically. That is, the following are equivalent inside a multiclass:

```
defm Foo      : SomeMultiClass<...>;
defm NAME#Foo : SomeMultiClass<...>;
```

See Interlude 2: Multiclass and Defm Examples for examples.

1.6.7 Interlude 2: Multiclass and defm examples

Here is a simple example using `multiclass` and `defm`. Consider a 3-address instruction architecture whose instructions come in two formats: `reg = reg op reg` and `reg = reg op imm` (immediate). The SPARC is an example of such an architecture.

```
def ops;
def GPR;
def Imm;
class inst <int opc, string asmstr, dag operandlist>;

multiclass ri_inst <int opc, string asmstr> {
  def _rr : inst<opc, !strconcat(asmstr, " $dst, $src1, $src2"),
                   (ops GPR:$dst, GPR:$src1, GPR:$src2)>;
  def _ri : inst<opc, !strconcat(asmstr, " $dst, $src1, $src2"),
                   (ops GPR:$dst, GPR:$src1, Imm:$src2)>;
}

// Define records for each instruction in the RR and RI formats.
defm ADD : ri_inst<0b111, "add">;
defm SUB : ri_inst<0b101, "sub">;
defm MUL : ri_inst<0b100, "mul">;
```

Each use of the `ri_inst` multiclass defines two records, one with the `_rr` suffix and one with `_ri`. Recall that the name of the `defm` that uses a multiclass is prepended to the names of the records defined in that multiclass. So the resulting definitions are named:

```
ADD_rr, ADD_ri
SUB_rr, SUB_ri
MUL_rr, MUL_ri
```

Without the `multiclass` feature, the instructions would have to be defined as follows.

```
def ops;
def GPR;
def Imm;
class inst <int opc, string asmstr, dag operandlist>;

class rrinst <int opc, string asmstr>
  : inst<opc, !strconcat(asmstr, " $dst, $src1, $src2"),
           (ops GPR:$dst, GPR:$src1, GPR:$src2)>;
```

```
class riinst <int opc, string asmstr>
  : inst<opc, !strconcat(asmstr, " $dst, $src1, $src2"),
           (ops GPR:$dst, GPR:$src1, Imm:$src2)>;

// Define records for each instruction in the RR and RI formats.
def ADD_rr : rrinst<0b111, "add">;
def ADD_ri : riinst<0b111, "add">;
def SUB_rr : rrinst<0b101, "sub">;
def SUB_ri : riinst<0b101, "sub">;
def MUL_rr : rrinst<0b100, "mul">;
def MUL_ri : riinst<0b100, "mul">;
```

A `defm` can be used in a multiclass to gain multiple levels of record instantiations.

```
class Instruction <bits<4> opc, string Name> {
  bits<4> opcode = opc;
  string name = Name;
}

multiclass basic_r <bits<4> opc> {
  def rr : Instruction<opc, "rr">;
  def rm : Instruction<opc, "rm">;
}

multiclass basic_s <bits<4> opc> {
  defm SS : basic_r<opc>;
  defm SD : basic_r<opc>;
  def X : Instruction<opc, "x">;
}

multiclass basic_p <bits<4> opc> {
  defm PS : basic_r<opc>;
  defm PD : basic_r<opc>;
  def Y : Instruction<opc, "y">;
}

defm ADD : basic_s<0xf>, basic_p<0xf>;
```

The final `defm` creates the following records, five from the `basic_s` multiclass and five from the `basic_p` multiclass:

```
ADDSSrr, ADDSSrm
ADDSDrr, ADDSDrm
ADDX
ADDPSrr, ADDPSrm
ADDPDrr, ADDPDrm
ADDY
```

A `defm` statement, both at top level and in a multiclass, can inherit from regular classes in addition to multiclasses. The rule is that the regular classes must be listed after the multiclasses, and there must be at least one multiclass.

```
class XD {
  bits<4> Prefix = 11;
}
class XS {
  bits<4> Prefix = 12;
}
class I <bits<4> op> {
  bits<4> opcode = op;
}

multiclass R {
  def rr : I<4>;
  def rm : I<2>;
}

multiclass Y {
  defm SS : R, XD;    // First multiclass R, then regular class XD.
  defm SD : R, XS;
}

defm Instr : Y;
```

This example will create four records, shown here in alphabetical order with their fields.

```
def InstrSDrm {
  bits<4> opcode = { 0, 0, 1, 0 };
  bits<4> Prefix = { 1, 1, 0, 0 };
}

def InstrSDrr {
  bits<4> opcode = { 0, 1, 0, 0 };
  bits<4> Prefix = { 1, 1, 0, 0 };
}

def InstrSSrm {
  bits<4> opcode = { 0, 0, 1, 0 };
  bits<4> Prefix = { 1, 0, 1, 1 };
}

def InstrSSrr {
  bits<4> opcode = { 0, 1, 0, 0 };
  bits<4> Prefix = { 1, 0, 1, 1 };
}
```

It’s also possible to use `let` statements inside multiclasses, providing another way to factor out commonality from the records, especially when using several levels of multiclass instantiations.

```
multiclass basic_r <bits<4> opc> {
  let Predicates = [HasSSE2] in {
    def rr : Instruction<opc, "rr">;
    def rm : Instruction<opc, "rm">;
```

```
      }
      let Predicates = [HasSSE3] in
        def rx : Instruction<opc, "rx">;
    }

    multiclass basic_ss <bits<4> opc> {
      let IsDouble = 0 in
        defm SS : basic_r<opc>;

      let IsDouble = 1 in
        defm SD : basic_r<opc>;
    }

    defm ADD : basic_ss<0xf>;
```

1.6.8 `defset` — create a definition set

The `defset` statement is used to collect a set of records into a global list of records.

```
Defset ::=  "defset" Type TokIdentifier "=" "{" Statement* "}"
```

All records defined inside the braces via `def` and `defm` are defined as usual, and they are also collected in a global list of the given name (**`TokIdentifier`**).

The specified type must be `list<`*class*`>`, where *class* is some record class. The `defset` statement establishes a scope for its statements. It is an error to define a record in the scope of the `defset` that is not of type *class*.

The `defset` statement can be nested. The inner `defset` adds the records to its own set, and all those records are also added to the outer set.

Anonymous records created inside initialization expressions using the `ClassID<...>` syntax are not collected in the set.

1.6.9 `defvar` — define a variable

A `defvar` statement defines a global variable. Its value can be used throughout the statements that follow the definition.

```
Defvar ::=  "defvar" TokIdentifier "=" Value ";"
```

The identifier on the left of the `=` is defined to be a global variable whose value is given by the value expression on the right of the `=`. The type of the variable is automatically inferred.

Once a variable has been defined, it cannot be set to another value.

Variables defined in a top-level `foreach` go out of scope at the end of each loop iteration, so their value in one iteration is not available in the next iteration. The following `defvar` will not work:

```
defvar i = !add(i, 1)
```

Variables can also be defined with `defvar` in a record body. See Defvar in Record Body for more details.

1.6.10 `foreach` — iterate over a sequence

The `foreach` statement iterates over a series of statements, varying a variable over a sequence of values.

```
Foreach         ::=  "foreach" ForeachIterator "in" "{" Statement* "}"
                    | "foreach" ForeachIterator "in" Statement
ForeachIterator ::=  TokIdentifier "=" ("{" RangeList "}" | RangePiece | Value)
```

The body of the `foreach` is a series of statements in braces or a single statement with no braces. The statements are re-evaluated once for each value in the range list, range piece, or single value. On each iteration, the **`TokIdentifier`** variable is set to the value and can be used in the statements.

The statement list establishes an inner scope. Variables local to a `foreach` go out of scope at the end of each loop iteration, so their values do not carry over from one iteration to the next. Foreach loops may be nested.

The `foreach` statement can also be used in a record **`Body`**.

Here is a simple example.

```
foreach i = [0, 1, 2, 3] in {
  def R#i : Register<...>;
  def F#i : Register<...>;
}
```

This loop defines records named `R0`, `R1`, `R2`, and `R3`, along with `F0`, `F1`, `F2`, and `F3`.

1.6.11 `if` — select statements based on a test

The `if` statement allows one of two statement groups to be selected based on the value of an expression.

```
If     ::=  "if" Value "then" IfBody
           | "if" Value "then" IfBody "else" IfBody
IfBody ::=  "{" Statement* "}" | Statement
```

The value expression is evaluated. If it evaluates to true (in the same sense used by the bang operators), then the statements following the `then` reserved word are processed. Otherwise, if there is an `else` reserved word, the statements following the `else` are processed. If the value is false and there is no `else` arm, no statements are processed.

Because the braces around the `then` statements are optional, this grammar rule has the usual ambiguity with “dangling else” clauses, and it is resolved in the usual way: in a case like `if v1 then if v2 then {...} else {...}`, the `else` associates with the inner `if` rather than the outer one.

The **`IfBody`** of the then and else arms of the `if` establish an inner scope. Any `defvar` variables defined in the bodies go out of scope when the bodies are finished (see Defvar in Record Body for more details).

The `if` statement can also be used in a record **`Body`**.

1.7 Additional Details

1.7.1 Defvar in record body

In addition to defining global variables, the `defvar` statement can be used inside the **`Body`** of a class or record definition to define local variables. The scope of the variable extends from the `defvar` statement to the end of the body. It cannot be set to a different value within its scope. The `defvar` statement can also be used in the statement list of a `foreach`, which establishes a scope.

A variable named `V` in an inner scope shadows (hides) any variables `V` in outer scopes. In particular, `V` in a record body shadows a global `V`, and `V` in a `foreach` statement list shadows any `V` in surrounding global or record scopes.

Variables defined in a `foreach` go out of scope at the end of each loop iteration, so their value in one iteration is not available in the next iteration. The following `defvar` will not work:

```
defvar i = !add(i, 1)
```

1.7.2 How records are built

The following steps are taken by TableGen when a record is built. Classes are simply abstract records and so go through the same steps.

1. Build the record name (**`NameValue`**) and create an empty record.
2. Parse the superclasses from left to right, visiting each superclass’s parent classes in depth-first order.
   a. Add the fields from the superclass to the record.
   b. Substitute the template arguments into those fields.
   c. Add the superclass to the record’s list of inherited classes.
3. Apply any top-level `let` bindings to the record. Recall that top-level bindings only apply to inherited fields.
4. Parse the body of the record.
   - Add any fields to the record.
   - Modify the values of fields according to local `let` statements.
   - Define any `defvar` variables.

5. Make a pass over all the fields to resolve any inter-field references.
6. Add the record to the master record list.

Because references between fields are resolved (step 5) after `let` bindings are applied (step 3), the `let` statement has unusual power. For example:

```
class C <int x> {
  int Y = x;
  int Yplus1 = !add(Y, 1);
  int xplus1 = !add(x, 1);
}

let Y = 10 in {
  def rec1 : C<5> {
  }
}

def rec2 : C<5> {
  let Y = 10;
}
```

In both cases, one where a top-level `let` is used to bind `Y` and one where a local `let` does the same thing, the results are:

```
def rec1 {      // C
  int Y = 10;
  int Yplus1 = 11;
  int xplus1 = 6;
}
def rec2 {      // C
  int Y = 10;
  int Yplus1 = 11;
  int xplus1 = 6;
}
```

`Yplus1` is 11 because the `let Y` is performed before the `!add(Y, 1)` is resolved. Use this power wisely.

1.8 Preprocessing Facilities

The preprocessor embedded in TableGen is intended only for simple conditional compilation. It supports the following directives, which are specified somewhat informally.

```
LineBegin              ::=  beginning of line
LineEnd                ::=  newline | return | EOF
WhiteSpace             ::=  space | tab
CComment               ::=  "/*" ... "*/"
BCPLComment            ::=  "//" ... LineEnd
WhiteSpaceOrCComment   ::=  WhiteSpace | CComment
WhiteSpaceOrAnyComment ::=  WhiteSpace | CComment | BCPLComment
```

```
MacroName               ::=  ualpha (ualpha | "0"..."9")*
PreDefine               ::=  LineBegin (WhiteSpaceOrCComment)*
                             "#define" (WhiteSpace)+ MacroName
                             (WhiteSpaceOrAnyComment)* LineEnd
PreIfdef                ::=  LineBegin (WhiteSpaceOrCComment)*
                             ("#ifdef" | "#ifndef") (WhiteSpace)+ MacroName
                             (WhiteSpaceOrAnyComment)* LineEnd
PreElse                 ::=  LineBegin (WhiteSpaceOrCComment)*
                             "#else" (WhiteSpaceOrAnyComment)* LineEnd
PreEndif                ::=  LineBegin (WhiteSpaceOrCComment)*
                             "#endif" (WhiteSpaceOrAnyComment)* LineEnd
```

A `MacroName` can be defined anywhere in a TableGen file. The name has no value; it can only be tested to see whether it is defined.

A macro test region begins with an `#ifdef` or `#ifndef` directive. If the macro name is defined (`#ifdef`) or undefined (`#ifndef`), then the source code between the directive and the corresponding `#else` or `#endif` is processed. If the test fails but there is an `#else` portion, the source code between the `#else` and the `#endif` is processed. If the test fails and there is no `#else` portion, then no source code in the test region is processed.

Test regions may be nested, but they must be properly nested. A region started in a file must end in that file; that is, must have its `#endif` in the same file.

A `MacroName` may be defined externally using the `-D` option on the `llvm-tblgen` command line:

```
llvm-tblgen self-reference.td -Dmacro1 -Dmacro3
```

1.9 Appendix A: Bang Operators

Bang operators act as functions in value expressions. A bang operator takes one or more arguments, operates on them, and produces a result. If the operator produces a boolean result, the result value will be 1 for true or 0 for false. When an operator tests a boolean argument, it interprets 0 as false and non-0 as true.

`!add(`*a*`,` *b*`, ...)`
  This operator adds *a*, *b*, etc., and produces the sum.

`!and(`*a*`,` *b*`, ...)`
  This operator does a bitwise AND on *a*, *b*, etc., and produces the result.

`!cast<`*type*`>(`*a*`)`
  This operator performs a cast on *a* and produces the result. If *a* is not a string, then a straightforward cast is performed, say between an `int` and a `bit`, or between record types. This allows casting a record to a class. If a record is cast to `string`, the record's name is produced.

  If *a* is a string, then it is treated as a record name and looked up in the list of all defined records. The resulting record is expected to be of the specified *type*.

For example, if `!cast<`*type*`>(`*name*`)` appears in a multiclass definition, or in a class instantiated inside a multiclass definition, and the *name* does not reference any template arguments of the multiclass, then a record by that name must have been instantiated earlier in the source file. If *name* does reference a template argument, then the lookup is delayed until `defm` statements instantiating the multiclass (or later, if the defm occurs in another multiclass and template arguments of the inner multiclass that are referenced by *name* are substituted by values that themselves contain references to template arguments of the outer multiclass).

If the type of *a* does not match *type*, TableGen raises an error.

`!con(`*a*`, `*b*`, ...)`
This operator concatenates the DAG nodes *a*, *b*, etc. Their operations must equal.

`!con((op a1:$name1, a2:$name2), (op b1:$name3))`

results in the DAG node `(op a1:$name1, a2:$name2, b1:$name3)`.

`!cond(`*cond1* `:` *val1*`, `*cond2* `:` *val2*`, ..., `*condn* `:` *valn*`)`
This operator tests *cond1* and returns *val1* if the result is true. If false, the operator tests *cond2* and returns *val2* if the result is true. And so forth. An error is reported if no conditions are true.

This example produces the sign word for an integer:

```
!cond(!lt(x, 0) : "negative", !eq(x, 0) : "zero", 1 : "positive")
```

`!dag(`*op*`, `*children*`, `*names*`)`
This operator creates a DAG node. The *children* and *names* arguments must be lists of equal length or uninitialized (`?`). The *names* argument must be of type `list<string>`.

Due to limitations of the type system, *children* must be a list of items of a common type. In practice, this means that they should either have the same type or be records with a common superclass. Mixing `dag` and non-`dag` items is not possible. However, `?` can be used.

Example: `!dag(op, [a1, a2, ?], ["name1", "name2", "name3"])` results in `(op a1:$name1, a2:$name2, ?:$name3)`.

`!empty(`*list*`)`
This operator produces 1 if the *list* is empty; 0 otherwise.

`!eq( `*a*`, `*b*`)`
This operator produces 1 if *a* is equal to *b*; 0 otherwise. The arguments must be `bit`, `int`, or `string` values. Use `!cast<string>` to compare other types of objects.

`!foldl(`*start*`, `*list*`, `*a*`, `*b*`, `*expr*`)`
This operator performs a left-fold over the items in *list*. The variable *a* acts as the accumulator and is initialized to *start*. The variable *b* is bound to each element in the *list*. The *expr* expression is evaluated

for each element and presumably uses *a* and *b* to calculate the accumulated value, which `!foldl` stores in *a*. The type of *a* is the same as *start*; the type of *b* is the same as the elements of *list*; *expr* must have the same type as *start*.

The following example computes the total of the `Number` field in the list of records in `RecList`:

```
int x = !foldl(0, RecList, total, rec, !add(total, rec.Number));
```

`!foreach(`*var*`,` *seq*`,` *form*`)`

This operator creates a new `list`/`dag` in which each element is a function of the corresponding element in the *seq* `list`/`dag`. To perform the function, TableGen binds the variable *var* to an element and then evaluates the *form* expression. The form presumably refers to the variable *var* and calculates the result value.

`!ge(`*a*, *b*`)`

This operator produces 1 if *a* is greater than or equal to *b*; 0 otherwise. The arguments must be `bit`, `int`, or `string` values. Use `!cast<string>` to compare other types of objects.

`!getop(`*dag*`)` –or– `!getop<`*type*`>(`*dag*`)`

This operator produces the operator of the given *dag* node. Example: `!getop((foo 1, 2))` results in `foo`.

The result of `!getop` can be used directly in a context where any record value at all is acceptable (typically placing it into another dag value). But in other contexts, it must be explicitly cast to a particular class type. The <*type*> syntax is provided to make this easy.

For example, to assign the result to a value of type `BaseClass`, you could write either of these:

```
BaseClass b = !getop<BaseClass>(someDag);
BaseClass b = !cast<BaseClass>(!getop(someDag));
```

But to create a new DAG node that reuses the operator from another, no cast is necessary:

```
dag d = !dag(!getop(someDag), args, names);
```

`!gt(`*a*, *b*`)`

This operator produces 1 if *a* is greater than *b*; 0 otherwise. The arguments must be `bit`, `int`, or `string` values. Use `!cast<string>` to compare other types of objects.

`!head(`*a*`)`

This operator produces the zeroth element of the list *a*. (See also `!tail`.)

`!if(`*test*`,` *then*`,` *else*`)`

This operator evaluates the *test*, which must produce a `bit` or `int`. If the result is not 0, the *then* expression is produced; otherwise the *else* expression is produced.

`!isa<`*type*`>(`*a*`)`

This operator produces 1 if the type of *a* is a subtype of the given *type*; 0 otherwise.

`!le(`*a*`,` *b*`)`
: This operator produces 1 if *a* is less than or equal to *b*; 0 otherwise. The arguments must be `bit`, `int`, or `string` values. Use `!cast<string>` to compare other types of objects.

`!listconcat(`*list1*`,` *list2*`, ...)`
: This operator concatenates the list arguments *list1*, *list2*, etc., and produces the resulting list. The lists must have the same element type.

`!listsplat(`*value*`,` *count*`)`
: This operator produces a list of length *count* whose elements are all equal to the *value*. For example, `!listsplat(42, 3)` results in `[42, 42, 42]`.

`!lt(`*a*`,` *b*`)`
: This operator produces 1 if *a* is less than *b*; 0 otherwise. The arguments must be `bit`, `int`, or `string` values. Use `!cast<string>` to compare other types of objects.

`!mul(`*a*`,` *b*`, ...)`
: This operator multiplies *a*, *b*, etc., and produces the product.

`!ne(`*a*`,` *b*`)`
: This operator produces 1 if *a* is not equal to *b*; 0 otherwise. The arguments must be `bit`, `int`, or `string` values. Use `!cast<string>` to compare other types of objects.

`!or(`*a*`,` *b*`, ...)`
: This operator does a bitwise OR on *a*, *b*, etc., and produces the result.

`!setop(`*dag*`,` *op*`)`
: This operator produces a DAG node with the same arguments as *dag*, but with its operator replaced with *op*.

  Example: `!setop((foo 1, 2), bar)` results in `(bar 1, 2)`.

`!shl(`*a*`,` *count*`)`
: This operator shifts *a* left logically by *count* bits and produces the resulting value. The operation is performed on a 64-bit integer; the result is undefined for shift counts outside 0..63.

`!size(`*a*`)`
: This operator produces the number of elements in the list *a*.

`!sra(`*a*`,` *count*`)`
: This operator shifts *a* right arithmetically by *count* bits and produces the resulting value. The operation is performed on a 64-bit integer; the result is undefined for shift counts outside 0..63.

`!srl(`*a*`,` *count*`)`
: This operator shifts *a* right logically by *count* bits and produces the resulting value. The operation is performed on a 64-bit integer; the result is undefined for shift counts outside 0..63.

`!strconcat(`*str1*`, `*str2*`, ...)`

This operator concatenates the string arguments *str1*, *str2*, etc., and produces the resulting string.

str1`#`*str2*

The paste operator (`#`) is a shorthand for `!strconcat` with two arguments. It can be used to concatenate operands that are not strings, in which case an implicit `!cast<string>` is done on those operands.

`!subst(`*target*`, `*repl*`, `*value*`)`

This operator replaces all occurrences of the *target* in the *value* with the *repl* and produces the resulting value. For strings, this is straightforward.

If the arguments are record names, the function produces the *repl* record if the *target* record name equals the *value* record name; otherwise it produces the *value*.

`!tail(`*a*`)`

This operator produces a new list with all the elements of the list *a* except for the zeroth one. (See also `!head`.)

1.10 Appendix B: Sample Record

The Intel x86 processors are complex. The following output from TableGen shows the record that is created to represent the 32-bit register-to-register ADD instruction.

```
def ADD32rr { // InstructionEncoding Instruction X86Inst I ITy Sched BinOpRR BinOpRR_RF
  int Size = 0;
  string DecoderNamespace = "";
  list<Predicate> Predicates = [];
  string DecoderMethod = "";
  bit hasCompleteDecoder = 1;
  string Namespace = "X86";
  dag OutOperandList = (outs GR32:$dst);
  dag InOperandList = (ins GR32:$src1, GR32:$src2);
  string AsmString = "add{l}  {$src2, $src1|$src1, $src2}";
  EncodingByHwMode EncodingInfos = ?;
  list<dag> Pattern = [(set GR32:$dst, EFLAGS, (X86add_flag GR32:$src1, GR32:$src2))];
  list<Register> Uses = [];
  list<Register> Defs = [EFLAGS];
  int CodeSize = 3;
  int AddedComplexity = 0;
  bit isPreISelOpcode = 0;
  bit isReturn = 0;
  bit isBranch = 0;
  bit isEHScopeReturn = 0;
  bit isIndirectBranch = 0;
  bit isCompare = 0;
  bit isMoveImm = 0;
  bit isMoveReg = 0;
  bit isBitcast = 0;
bit isSelect = 0;
bit isBarrier = 0;
bit isCall = 0;
bit isAdd = 0;
bit isTrap = 0;
bit canFoldAsLoad = 0;
bit mayLoad = ?;
bit mayStore = ?;
bit mayRaiseFPException = 0;
bit isConvertibleToThreeAddress = 1;
bit isCommutable = 1;
bit isTerminator = 0;
bit isReMaterializable = 0;
bit isPredicable = 0;
bit isUnpredicable = 0;
bit hasDelaySlot = 0;
bit usesCustomInserter = 0;
bit hasPostISelHook = 0;
bit hasCtrlDep = 0;
bit isNotDuplicable = 0;
bit isConvergent = 0;
bit isAuthenticated = 0;
bit isAsCheapAsAMove = 0;
bit hasExtraSrcRegAllocReq = 0;
bit hasExtraDefRegAllocReq = 0;
bit isRegSequence = 0;
bit isPseudo = 0;
bit isExtractSubreg = 0;
bit isInsertSubreg = 0;
bit variadicOpsAreDefs = 0;
bit hasSideEffects = ?;
bit isCodeGenOnly = 0;
bit isAsmParserOnly = 0;
bit hasNoSchedulingInfo = 0;
InstrItinClass Itinerary = NoItinerary;
list<SchedReadWrite> SchedRW = [WriteALU];
string Constraints = "$src1 = $dst";
string DisableEncoding = "";
string PostEncoderMethod = "";
bits<64> TSFlags = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
string AsmMatchConverter = "";
string TwoOperandAliasConstraint = "";
string AsmVariantName = "";
bit UseNamedOperandTable = 0;
bit FastISelShouldIgnore = 0;
bits<8> Opcode = { 0, 0, 0, 0, 0, 0, 0, 1 };
Format Form = MRMDestReg;
bits<7> FormBits = { 0, 1, 0, 1, 0, 0, 0 };
ImmType ImmT = NoImm;
bit ForceDisassemble = 0;
OperandSize OpSize = OpSize32;
bits<2> OpSizeBits = { 1, 0 };
AddressSize AdSize = AdSizeX;
```

```
    bits<2> AdSizeBits = { 0, 0 };
    Prefix OpPrefix = NoPrfx;
    bits<3> OpPrefixBits = { 0, 0, 0 };
    Map OpMap = OB;
    bits<3> OpMapBits = { 0, 0, 0 };
    bit hasREX_WPrefix = 0;
    FPFormat FPForm = NotFP;
    bit hasLockPrefix = 0;
    Domain ExeDomain = GenericDomain;
    bit hasREPPrefix = 0;
    Encoding OpEnc = EncNormal;
    bits<2> OpEncBits = { 0, 0 };
    bit HasVEX_W = 0;
    bit IgnoresVEX_W = 0;
    bit EVEX_W1_VEX_W0 = 0;
    bit hasVEX_4V = 0;
    bit hasVEX_L = 0;
    bit ignoresVEX_L = 0;
    bit hasEVEX_K = 0;
    bit hasEVEX_Z = 0;
    bit hasEVEX_L2 = 0;
    bit hasEVEX_B = 0;
    bits<3> CD8_Form = { 0, 0, 0 };
    int CD8_EltSize = 0;
    bit hasEVEX_RC = 0;
    bit hasNoTrackPrefix = 0;
    bits<7> VectSize = { 0, 0, 1, 0, 0, 0, 0 };
    bits<7> CD8_Scale = { 0, 0, 0, 0, 0, 0, 0 };
    string FoldGenRegForm = ?;
    string EVEX2VEXOverride = ?;
    bit isMemoryFoldable = 1;
    bit notEVEX2VEXConvertible = 0;
  }
```

On the first line of the record, you can see that the `ADD32rr` record inherited from eight classes. Although the inheritance hierarchy is complex, using superclasses is much simpler than specifying the 109 individual fields for each instruction.

Here is the code fragment used to define `ADD32rr` and multiple other `ADD` instructions:

```
defm ADD : ArithBinOp_RF<0x00, 0x02, 0x04, "add", MRM0r, MRM0m,
                         X86add_flag, add, 1, 1, 1>;
```

The `defm` statement tells TableGen that `ArithBinOp_RF` is a multiclass, which contains multiple concrete record definitions that inherit from `BinOpRR_RF`. That class, in turn, inherits from `BinOpRR`, which inherits from `ITy` and `Sched`, and so forth. The fields are inherited from all the parent classes; for example, `IsIndirectBranch` is inherited from the `Instruction` class.